

Software-Projekt I

Prof. Dr. Rainer Koschke

Arbeitsgruppe Softwaretechnik
Fachbereich Mathematik und Informatik
Universität Bremen

Sommersemester 2013

Architekturstile und Entwurfsmuster I

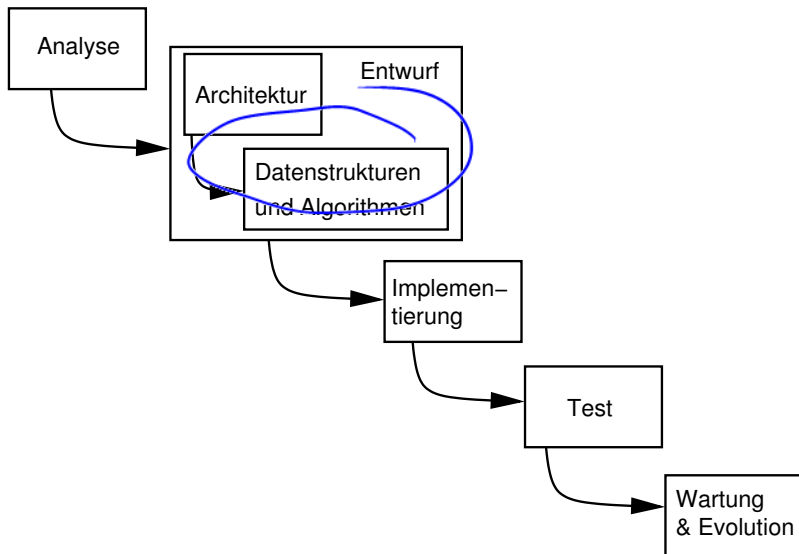
1 Architekturstile und Entwurfsmuster

- Was ist ein Entwurfsmuster?
- Bestandteile eines Entwurfsmusters
- Entwurfsmuster Singleton
- Kategorien von Entwurfsmustern
- Entwurfsmuster Command
- Entwurfsmuster Composite
- Entwurfsmuster Observer
- Architekturstil Model-View-Controller
- Architekturstil Schichtung
- Entwurfsmuster Memento
- Entwurfsmuster Factory Method



- Muss man immer alles neu entwerfen bzw. wie kann man auf bewährte Entwurfslösungen zurückgreifen?
- Wie können Entwurfsmuster und Architekturstile beim Entwurf helfen?
- Welche Eigenschaften haben Entwurfsmuster und Architekturstile?

Kontext



Entwurfsmuster

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander (Architekt und Mathematiker),
“A pattern language”, 1977.

Entwurfsmuster

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Christopher Alexander (Architekt und Mathematiker),
“A pattern language”, 1977.

Definition

Entwurfsmuster: „Musterlösung“ für ein wiederkehrendes Entwurfsproblem.

Bestandteile eines Entwurfsmusters

- **Name** (kurz und beschreibend)
- **Problem**: Was das Entwurfsmuster löst
- **Lösung**: Wie es das Problem löst
- **Konsequenzen**: Folgen und Kompromisse des Musters

Fragen



In Bibi soll es nur eine global verfügbare Repräsentation der Bibliothek (= Menge der Bücher) geben.

Das heißt: Bibliothek wird durch eine Klasse realisiert.
Davon darf es nur eine Instanz geben.

Wie lässt sich das erreichen?

Entwurfsmuster *Singleton*

- **Name** *Singleton*
- **Problem**: Stellt sicher, dass es von einer Klasse nur eine Instanz gibt.
- **Lösung**:

- ▶ Konstruktor privat machen
- ▶ statisches Attribut speichert einzige Instanz
- ▶ statische Methode liefert einzige Instanz zurück

Singleton
- static instance: Singleton - static ...
+ static Singleton getInstance() - Singleton() + ...

- **Konsequenzen**:
 - ▶ kontrollierter Zugriff auf einzige Instanz
 - ▶ (vermeidet globale Variablen)
 - ▶ verschleiert Abhängigkeiten
 - ▶ Testen wird erschwert
 - ▶ Einzigartigkeit in verteilten Umgebungen nicht sichergestellt

Entwurfsmuster *Singleton*

// the library: a list of books

```
public class Library {
```

```
    private Library () { super();} // hide constructor
```



// holds the list of books in the library

```
    private List<Book> books = new LinkedList<Book>();
```

// holds the only instance

```
    private static Library instance = new Library();
```

// returns the unique library instance

```
    public static Library getInstance () { return instance; }
```



// returns the book with the given author and title

```
    public Book get(String author, String title) {...}
```

```
    ...
```

```
}
```

Entwurfsmuster *Singleton* im Allgemeinen

```
1 public final class Singleton {
2
3     private static Singleton instance;
4     // speichert einzige Instanz
5
6     private Singleton() {}
7     // kann von außerhalb nicht benutzt werden
8
9     // liefert einzige Instanz
10    public synchronized static Singleton getInstance() {
11        if (instance == null) { // lazy instantiation
12            instance = new Singleton();
13        }
14        return instance;
15    }
16 }
```

Kategorien von Entwurfsmustern

- Erzeugungsmuster
 - betreffen die Erzeugung von Objekten
 - Beispiel: *Singleton*
- Strukturelle Muster:
 - betreffen Komposition von Klassen und Objekten
 - Beispiel: *Composite*
- Verhaltensmuster:
 - betreffen Interaktion und Verantwortlichkeiten
 - Beispiel: *Command*, *Memento*, *Observer*

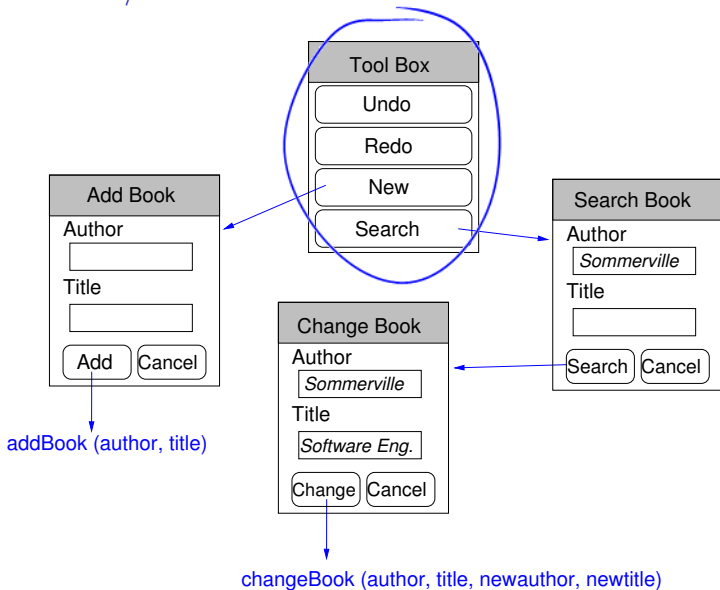


Wie kann man ein Undo/Redo implementieren?

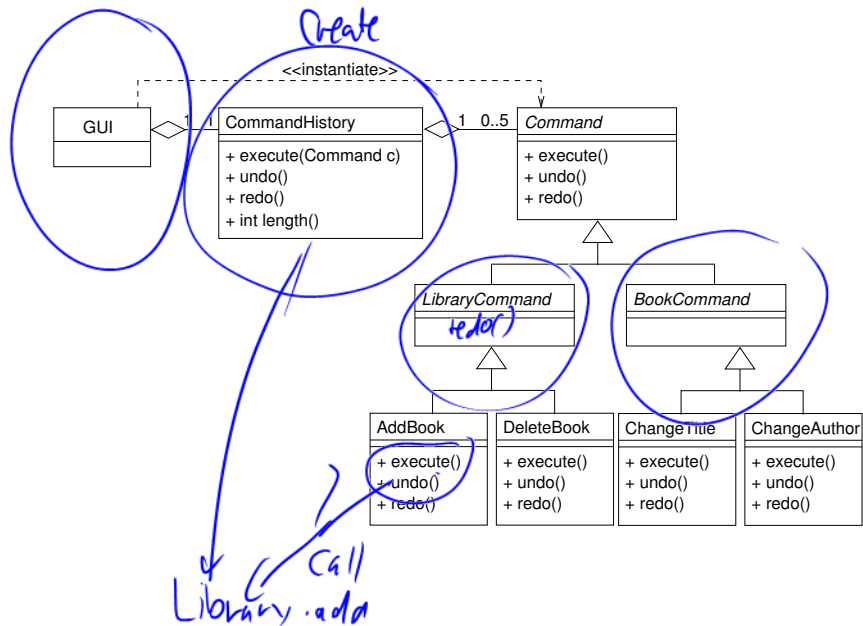
Benötigt:

- ① Aufzeichnung aller ausgeführten Benutzeraktionen mit Argumenten
- ② Umkehraktion für jede mögliche Aktion

GUI mit Undo/Redo



Entwurfsmuster Command



CommandHistory

```
1 // a bounded history of commands
2 public class CommandHistory {
3     // upper bound of commands
4     private static final int maxCommands = 5;
5     // saved commands
6     private Command[] commands = new Command [maxCommands];
7     // the current command in commands
8     private int current = -1;
9
10    // adds command to the history and executes it
11    public void execute (Command command) {
12        current++;
13        // in case of a full history, we need to shift
14        if (current == maxCommands) { // full? => shift
15            for (int i = 0; i < maxCommands - 1; i++)
16                commands[i] = commands[i+1];
17            current = maxCommands - 1;
18        }
19        commands [current] = command;
20        commands [current].execute();
21    }
```


CommandHistory (Forts.)

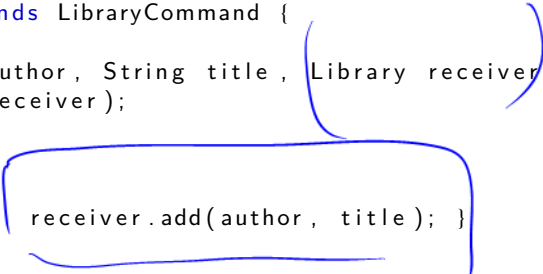
```
1  // undo for current command
2  public void undo () throws EmptyHistory , InvalidUndo {
3      commands [current].undo ();
4      current--;
5  }
6  // redo for current command
7  public void redo () throws EmptyHistory , InvalidRedo {
8      current++;
9      commands [current].redo ();
10 }
11 }
```

Command

```
1 // super class of all commands
2 public abstract class Command {
3     public abstract void execute ();
4     public abstract void undo ();
5     public abstract void redo ();
6 }
7
8 // super class of all commands that manipulate the library
9 public abstract class LibraryCommand extends Command {
10     protected String author; // library command argument
11     protected String title; // library command argument
12     protected Library receiver; // command receiver
13     protected void init(String author, String title,
14                          Library receiver) {
15         this.author = author;
16         this.title = title;
17         this.receiver = receiver;
18     }
19     @Override
20     public void redo() { execute(); }
21 }
```

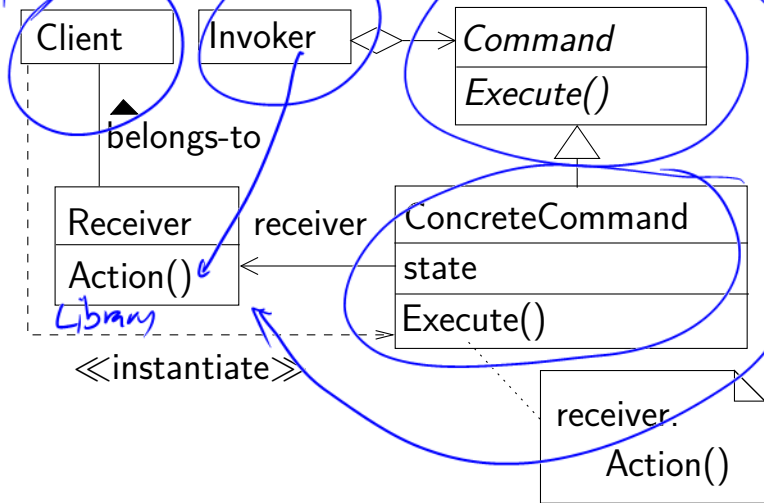
Command

```
1 // command to add a new book to the library
2 public class AddBook extends LibraryCommand {
3
4     public AddBook(String author, String title, Library receiver)
5         init(author, title, receiver);
6     }
7
8     @Override
9     public void execute() { receiver.add(author, title); }
10
11    @Override
12    public void undo() { receiver.delete(author, title); }
13 }
```

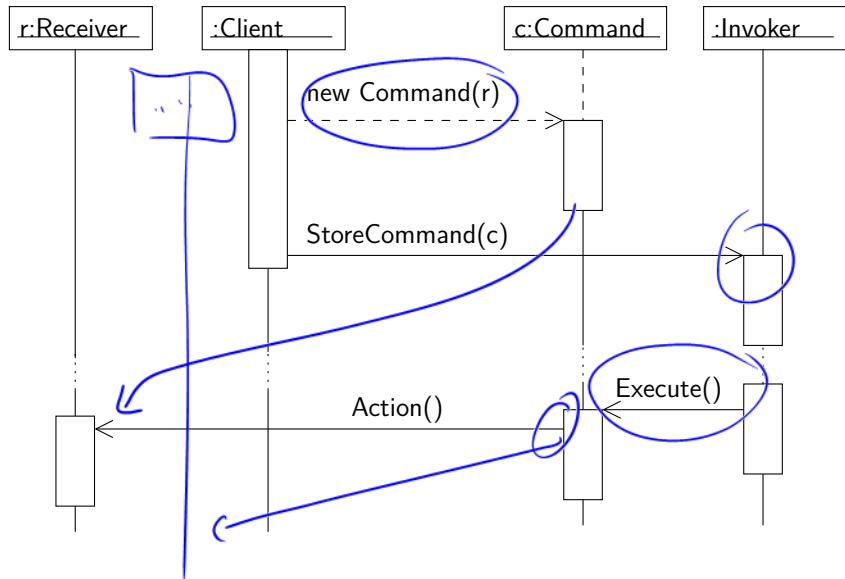


Command im Allgemeinen (command + listen)

GUI



Command im Allgemeinen: Interaktion



Konsequenzen

- Entkopplung von Objekt, das Operation aufruft, von dem, welches weiß, wie man es ausführt
- Kommandos sind selbst Objekte und können als solche verwendet werden (Attribute, Vererbung etc.)
- Hinzufügen weiterer Kommandos ist einfach

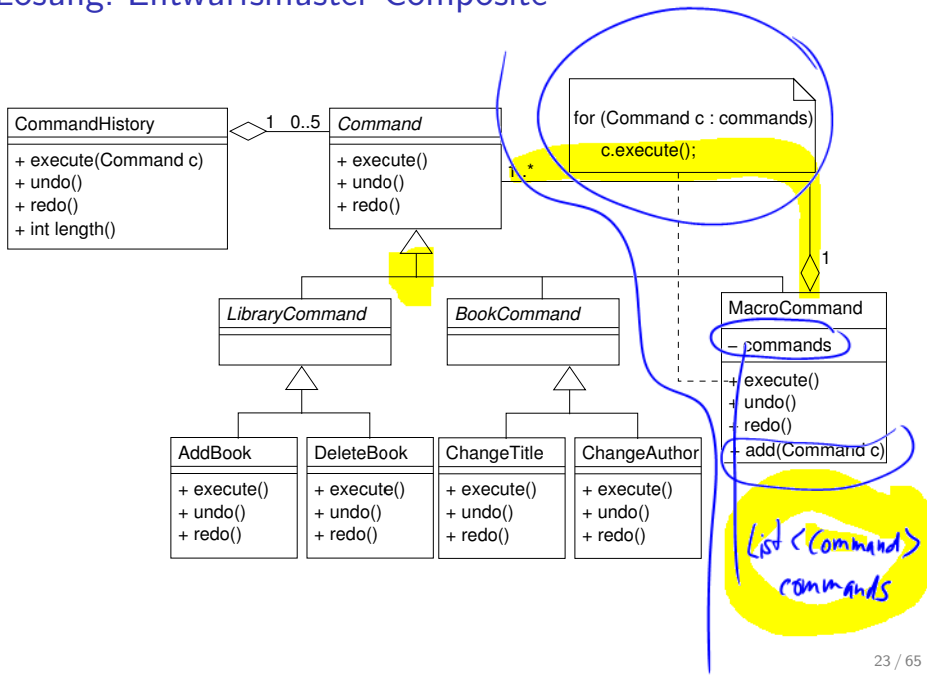


Kommandos sollen zu einer Script-Sprache ausgebaut werden. Weitere zusammengesetzte Kommandos (Makros, Schleifen, if-Anweisungen etc.) sollen eingeführt werden.

Die *CommandHistory* für Undo/Redo soll sich nicht mit dem Unterschied zu anderen Kommandos auseinander setzen müssen.

Wie lässt sich das erreichen?

Lösung: Entwurfsmuster Composite

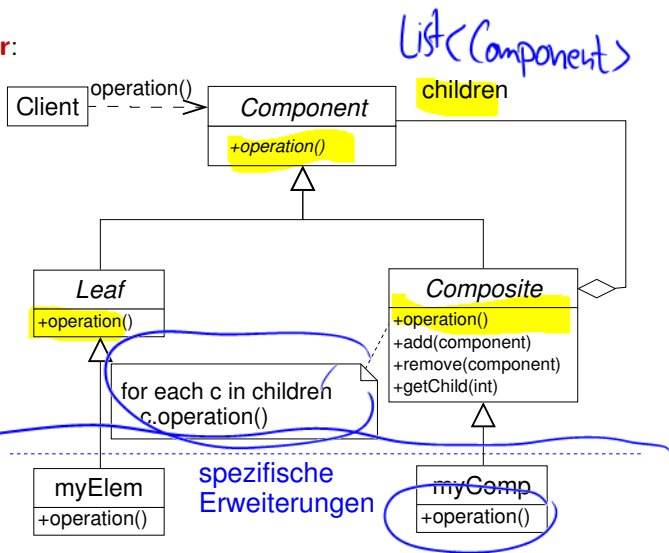


Entwurfsmuster Composite (Gamma u. a. 2003) I

- **Name:** *Composite*
- **Zweck:** Teil-von-Hierarchie mit einheitlicher Schnittstelle beschreiben (überall wo ein Ganzes benutzt werden kann, kann auch ein Teil benutzt werden und umgekehrt)

Entwurfsmuster Composite (Gamma u. a. 2003) II

- **Struktur:**




Entwurfsmuster Composite (Gamma u. a. 2003) I

Teilnehmer:

- *Client*:
 - ▶ manipuliert Objekte der Komponenten nur durch die Schnittstelle von *Composite*
- *Component*:
 - ▶ deklariert einheitliche Schnittstelle
 - ▶ (optional) implementiert Standardverhalten

```
public abstract class Component {  
    public abstract void operation ();  
}
```



Entwurfsmuster Composite (Gamma u. a. 2003) II

- *Leaf*:

- repräsentiert atomare Komponente
- definiert Verhalten für atomare Komponenten

```
public abstract class Leaf extends Component {  
    public abstract void operation ();  
}
```

Entwurfsmuster Composite (Gamma u. a. 2003) III

- *Composite*:

- ▶ definiert Standardverhalten für zusammengesetzte Komponenten
- ▶ speichert Teile
- ▶ implementiert Operationen zur Verwaltung von Teilen

```
import java.util.List;
import java.util.ArrayList;
public abstract class Composite extends Component {

    private List<Component> children
        = new ArrayList<Component>();

    public void operation () {
        for (Component c : children) c.operation ();
    }
    public void add (Component c) {children.add (c);}
    public void remove (Component c) {children.remove (c);}
    public Component getChild (int i) {return children.get (i);}
}
```

Entwurfsmuster Composite (Gamma u. a. 2003) I

Kollaborationen:

- *Clients* benutzen *Component*-Schnittstelle
- falls Empfänger ein *Leaf* ist, antwortet es direkt
- falls Empfänger ein *Composite* ist, wird die Anfrage an Teile weitergeleitet (möglicherweise mit weiteren eigenen Operationen vor und/oder nach der Weiterleitung)

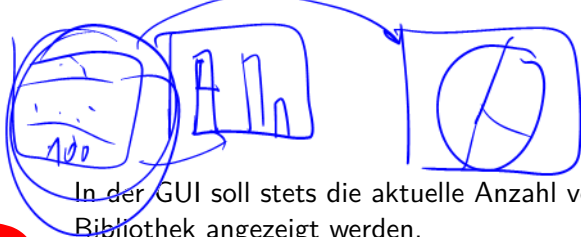
Entwurfsmuster Composite (Gamma u. a. 2003) II

Konsequenzen:

- zweiteilt die Klassenhierarchie in *Leaf* und *Composite* mit einheitlicher Schnittstelle
- uniforme Verwendung auf Seiten des *Clients*
- neue Komponenten können leicht hinzugefügt werden
- könnte die Struktur unnötig allgemein machen: nicht notwendigerweise jedes beliebige *Component* darf Teil eines *Composite* sein; Compiler kann das nicht überprüfen

Fragen

DB



In der GUI soll stets die aktuelle Anzahl von Büchern in der Bibliothek angezeigt werden.

Bei jedem Hinzufügen oder Löschen eines Buches muss die Anzeige angepasst werden.

Zukünftig könnte es noch viele weitere alternative Anzeigen geben (z.B. Balken statt Zahl).

Wie kann die Änderung eines Objektzustands einfach allen Interessenten mitgeteilt werden?

Entwurfsmuster Observer

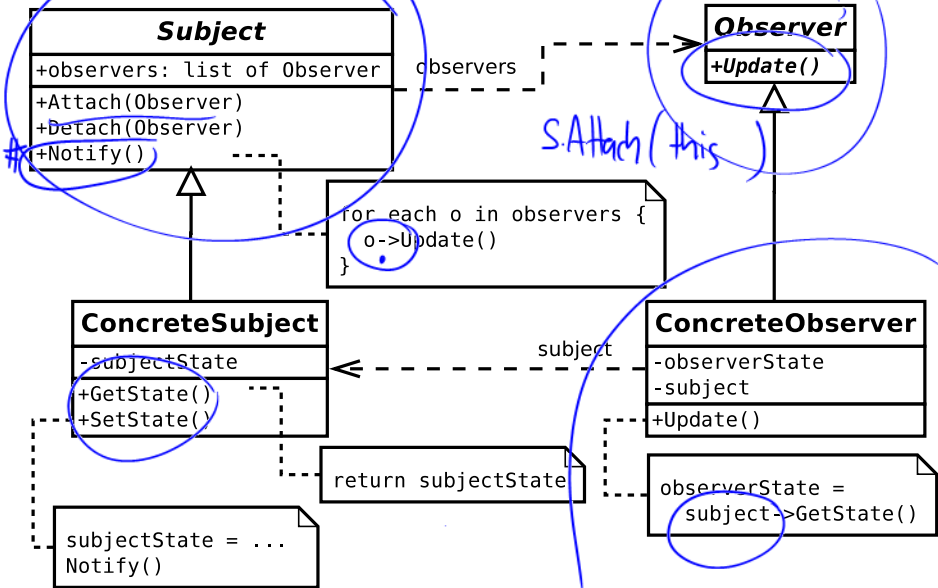
Anwendbarkeit

- Komponenten hängen von anderen Komponenten ab
- Änderung der einen Komponente muss Änderung der anderen nach sich ziehen
- Komponenten sollen lose gekoppelt sein: Komponente kennt seine Abhängigen nicht im Voraus (zur Übersetzungszeit)

Lösungsstrategie

- Abhängige registrieren sich bei Komponente
- Komponente informiert alle registrierten Abhängigen über Zustandsänderung

Entwurfsmuster Observer: Struktur



Entwurfsmuster Observer: Teilnehmer *Subject*

- kennt seine Observer (zur Laufzeit)
- kann beliebig viele Observer haben
- stellt Schnittstelle zur Verfügung, um Observer zu registrieren und abzutrennen

```
1 public interface Subject {  
2     public void Attach (Observer observer);  
3     public void Detach (Observer observer);  
4     public void Notify ();  
5 }
```

Entwurfsmuster Observer: Teilnehmer *Observer*

- deklariert Schnittstelle für die Update-Nachricht

```
1 public interface Observer {  
2     public void Update();  
3 }
```

Entwurfsmuster Observer: Teilnehmer *ConcreteSubject*

- implementiert Subject-Schnittstelle

```
1 public class Library implements Subject {  
2     ...  
3     private List<Observer> observers = new LinkedList<Observer>();  
4  
5     public void Attach(Observer observer) {  
6         observers.add(observer);  
7     }  
8     public void Detach(Observer observer) {  
9         observers.remove(observer);  
10    }  
11    public void Notify() {  
12        for (Observer o : observers)  
13            o.Update();  
14    }  
15 }
```

Entwurfsmuster Observer: Teilnehmer *ConcreteSubject*

- ruft bei jeder Zustandsänderung *Notify* auf

```
1 public class Library implements Subject {
2     ...
3
4     // adds a new new book with given author and title to the library
5     public void add (String author, String title) {
6         Book book = new Book(author, title);
7         books.add(book);
8         Notify();
9     }
10
11    // deletes a book with given author and title from the library
12    public void delete (String author, String title) {
13        Book book = get(author, title);
14        books.remove(book);
15        Notify();
16    }
17    ...
```

Entwurfsmuster Observer: Teilnehmer *ConcreteObserver*

- kennt ConcreteSubject-Objekt
- verarbeitet Zustand dieses Subjects
- implementiert Update, um auf veränderten Zustand zu reagieren

```
1 // an observer of the library
2 // emits the number of books upon every change
3 public class BookCounter implements Observer {
4
5     private Library subject; // the observer library
6
7     public BookCounter (Library subject) {
8         this.subject = subject;
9         subject.Attach (this);
10    }
11
12    // emit number of books upon every addition/removal
13    // of books to/from subject
14    public void Update () {
15        System.out.println ("number_of_books: "
16                             + subject.numberOfBooks());
17    }
18 }
```

Entwurfsmuster Observer: Konsequenzen

- abstrakte Kopplung zwischen Subject und Observer
- unterstützt Rundfunk (Broadcast)
- unerwartete Updates, komplizierter Kontrollfluss
- viel Nachrichtenverkehr, auch dann wenn sich ein irrelevanter Aspekt geändert hat

Entwurfsmuster Observer: Verfeinerungen

- Push-Modell

- ▶ Subject sendet detaillierte Beschreibung der Änderung
 - umfangreiches Update
 - vermeidet GetState(), aber nicht Update()

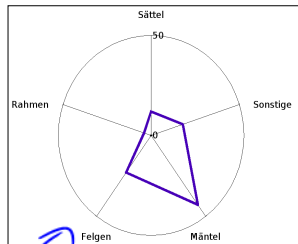
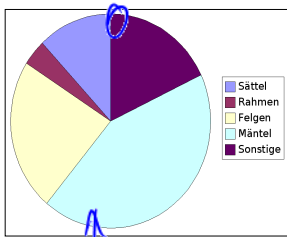
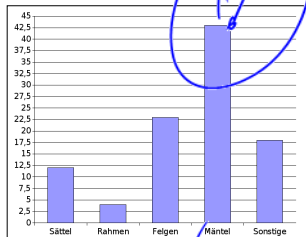
- Pull-Modell

- ▶ Subject sendet minimale Beschreibung der Änderung
 - Observer fragt gegebenenfalls die Details nach
 - erfordert weitere Nachrichten, um Details abzufragen

- Explizite Interessen

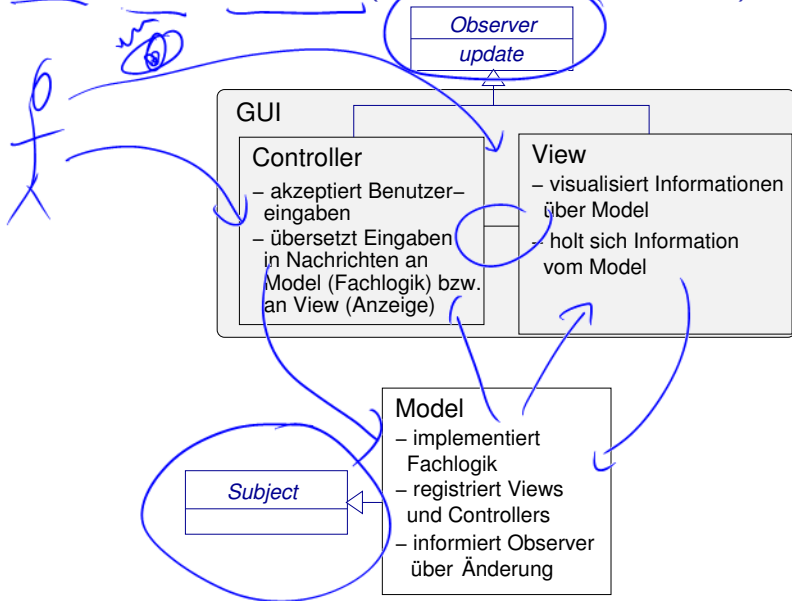
- ▶ Observers melden Interesse an spezifischem Aspekt an; Aspekt wird zusätzlicher Parameter von Update

Anforderungen

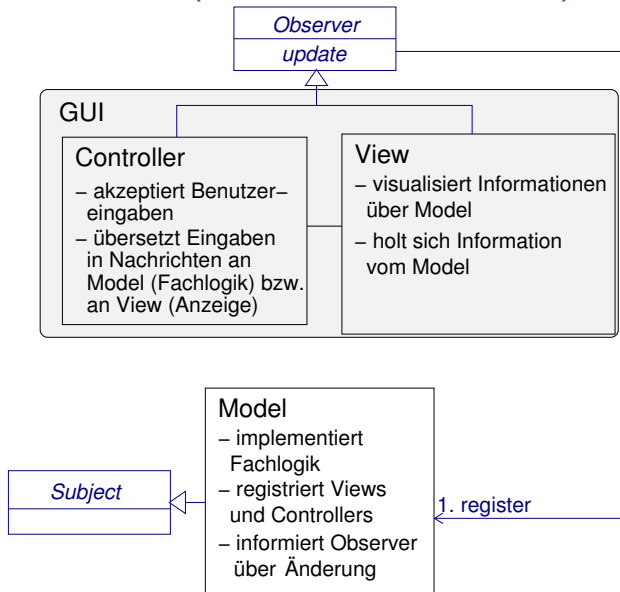


Sättel: 12%
Rahmen: 4%
Felgen: 23%
Mäntel: 43%
Sonstige: 18%

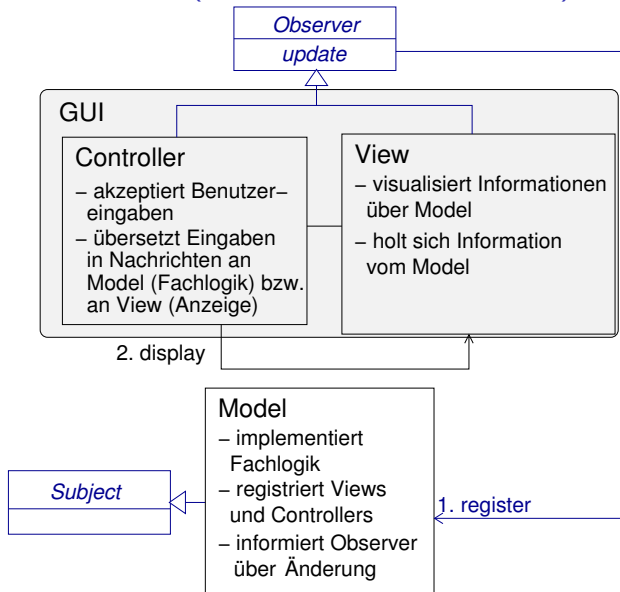
Model-View-Controller (Buschmann u. a. 1996)



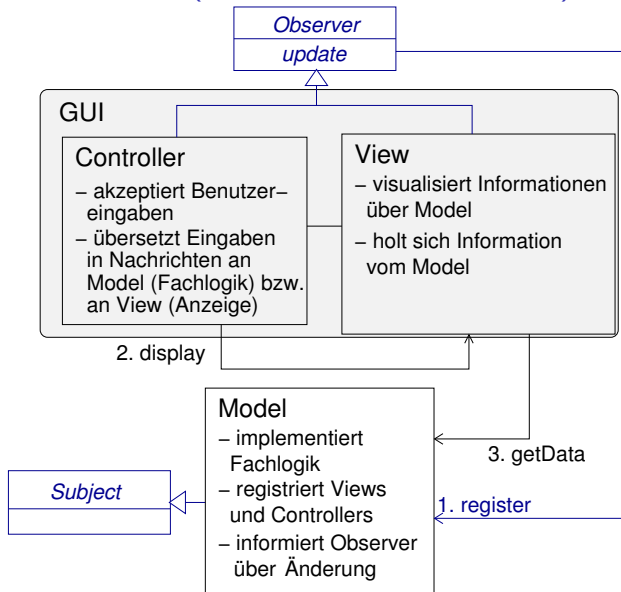
Model-View-Controller (Buschmann u. a. 1996)



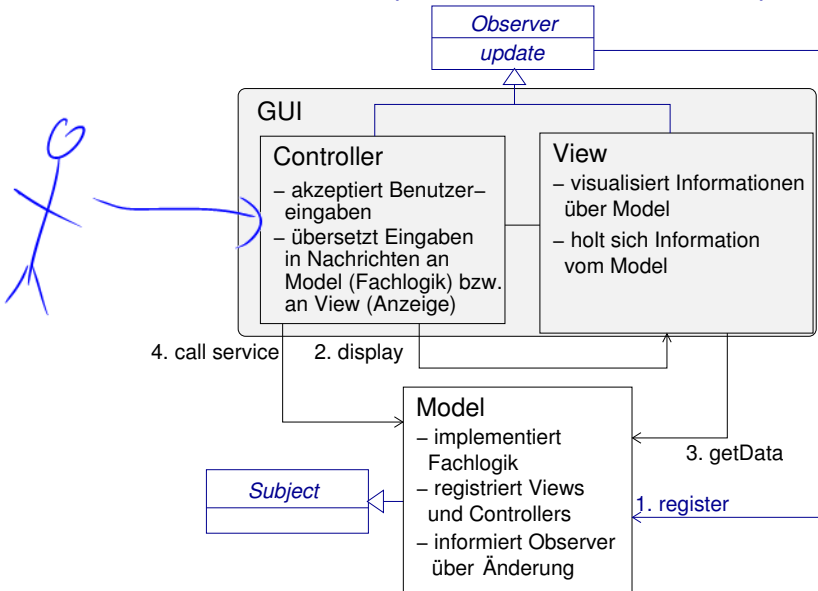
Model-View-Controller (Buschmann u. a. 1996)



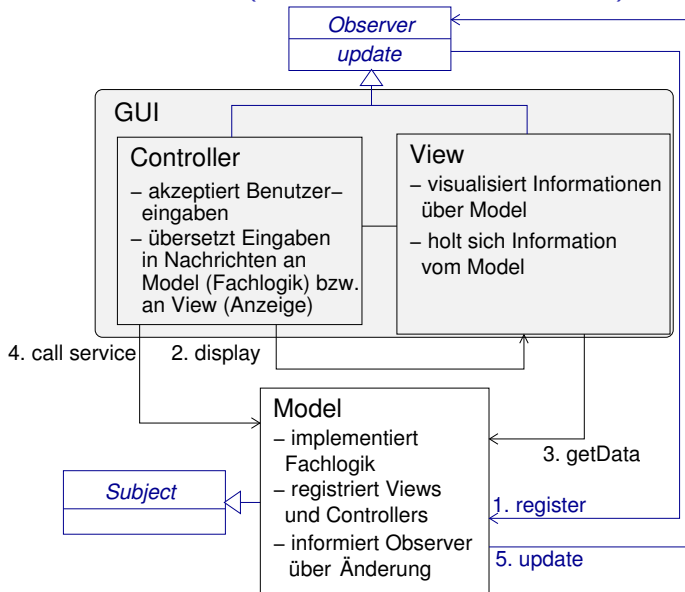
Model-View-Controller (Buschmann u. a. 1996)



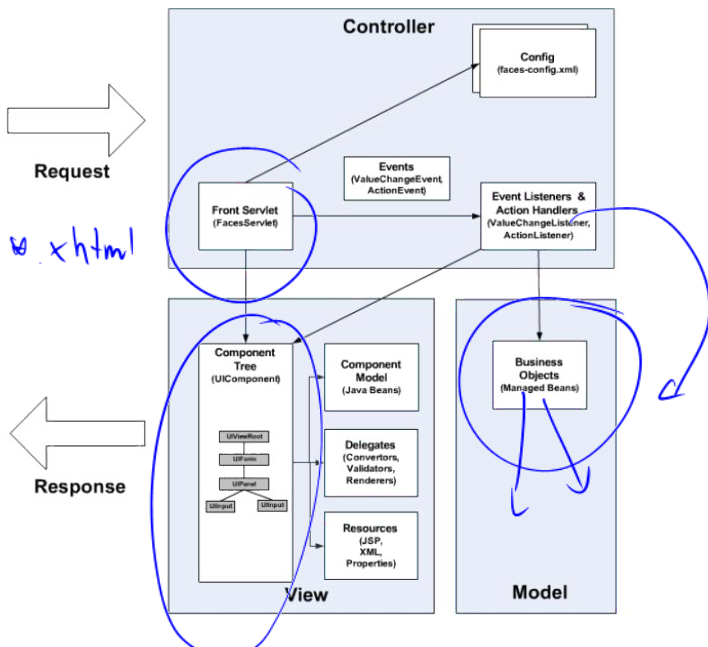
Model-View-Controller (Buschmann u. a. 1996)



Model-View-Controller (Buschmann u. a. 1996)



Model-View-Controller bei Java Server Faces



Liste der Bücher

Verliehen	Autor	Titel	ISBN
<input type="checkbox"/>	Rainer	That's it	123-12-423-1234-1
<input checked="" type="checkbox"/>	Dierk	That isn't it	123-12-423-1234-2

Anzahl ausgeliehener Bücher: 1

[Buch hinzufügen](#) [Leser anzeigen](#)

Facelet

```
<h:column>
  <f:facet name="header">
    <h:column>
      <h:outputText value="Verliehen"></h:outputText>
    </h:column>
  </f:facet>
```

```
<h:selectBooleanCheckbox
```

```
<!-- to retrieve the shown value -->
value="#{book.lent}"
```

```
<!-- submit event if changed -->
onchange="submit()"
```

```
<!-- notify frontPageBean -->
```

```
valueChangeListener="#{frontPageBean.checkBook}"
```

```
<!-- submit book as parameter -->
```

```
<f:attribute name="book" value="#{book}" />
```

```
</h:selectBooleanCheckbox>
```

```
</h:column>
```

```
...
```

```
<h:outputText value="Anzahl ausgeliehener Bücher: " />
```

```
<h:outputText value="#{frontPageBean.numberOfLentBooks}" />
```

Book

Book.getLent()

void setLent(Boolean b)

view

Bean

@ManagedBean

public class FrontPageBean {

public int getNumberOfLentBooks() {...}

public void checkBook(ValueChangeEvent e) {

Object newValue = e.get~~NewValue~~();

Object parameter =

((javax.faces.component.UIComponentBase)

e.getSource()).getAttributes().get("book");

if (parameter != null && parameter instanceof Book
&& newValue instanceof Boolean)

{
Book book = (Book) parameter;
book.setLent((Boolean) newValue);
BusinessHandler bh = BusinessHandler.getInstance();
bh.updateBook(book);
}
}

...

Architekturstil: Schichtung I

- Vokabular:
 - Komponenten: Module und Schichten
 - Konnektoren: Use-Beziehung
- Struktur:
 - Module sind eindeutig einer Schicht zugeordnet
 - Module einer Schicht dürfen nur auf Module derselben und der direkt darunter liegenden Schicht zugreifen
- Ausführungsmodell:
 - Aufruf von Methoden tieferer Schichten
 - Datenfluss in beide Richtungen (von der unteren Schicht zur oberen durch Rückgabeparameter)

Architekturstil: Schichtung II

- Vorteile:

- ▶ Schicht implementiert virtuelle Maschine, deren Implementierung leicht ausgetauscht werden kann, ohne dass höhere Schichten geändert werden müssen

- Nachteile:

- ▶ höherer Aufwand durch das „Durchreichen“ von Information
- ▶ Redundanz durch Dienste tieferer Schichten, die in hohen Schichten benutzt und auf allen Ebenen dazwischen repliziert werden

Anhang

Es folgen Muster, die eventuell in SWP-2 oder bei anderer Gelegenheit relevant werden könnten, die aber nicht Gegenstand der diesjährigen Vorlesung und der Prüfung sind.



Undo/Redo kennen den internen Zustand von *Book*.

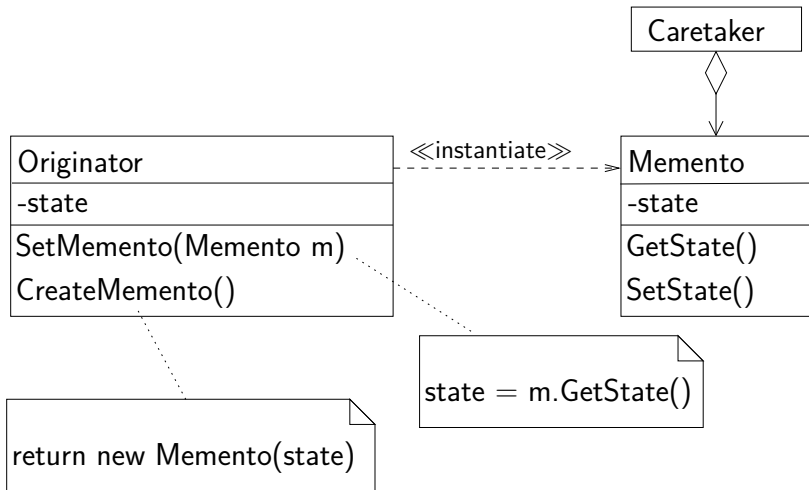
Wie lässt sich das vermeiden?

Entwurfsmuster *Memento*

Memento:

- ist Kopie des Zustands eines Ausgangsobjekts O
- wird von anderen Objekten benutzt, um früheren Zustand von O wieder herzustellen
- Zustand ist opak (undurchsichtig) für alle anderen Objekte

Memento im Allgemeinen



Memento für Undo/Redo

```
1 // a memento for a single book to save its state
2 public class BookMemento {
3     private String author; // state aspect of book
4     private String title;  // state aspect of book
5
6     // partial state accessors
7     public String getAuthor () { return author; }
8     public String getTitle () { return title; }
9
10    // memento state is set in constructor
11    public BookMemento (String author, String title)
12    { this.author = author;
13      this.title  = title;
14    }
15 }
```

Memento für Undo/Redo

```
1 // super class for all book commands;
2 // book commands use Memento design pattern for undo/redo
3 public abstract class BookCommand extends Command {
4
5     protected Book receiver;    // receiver of this command
6     private BookMemento memento; // state of the receiver book
7
8     BookCommand (Book receiver)
9     { this.receiver = receiver;
10       this.memento = receiver.createMemento();
11     }
12     @Override
13     public void redo() { execute (); }
14
15     @Override
16     public void undo() { receiver.setMemento(memento); }
17 }
```

Erweiterung von Bibi für andere Domänen

Anforderungen:

- Daten sollen von einer Datei gelesen werden können
- zukünftig sollen andere Domänen unterstützt werden (z.B. Videobibliothek)
- die Objekte dieser Domänen sind unterschiedlich
- notwendige Anpassungen sollen einfach vom Benutzer selbst realisiert werden können

Erweiterung von Bibi für andere Domänen

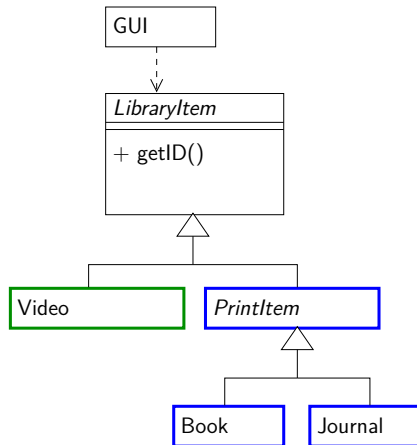
Anforderungen:

- Daten sollen von einer Datei gelesen werden können
- zukünftig sollen andere Domänen unterstützt werden (z.B. Videobibliothek)
- die Objekte dieser Domänen sind unterschiedlich
- notwendige Anpassungen sollen einfach vom Benutzer selbst realisiert werden können

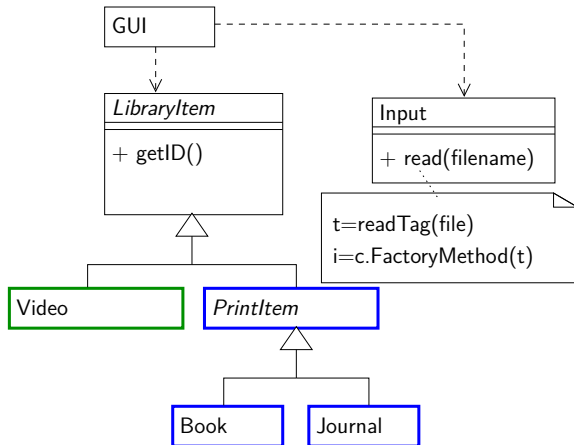
Lösungsstrategien:

- die Klassen der Benutzungsschnittstelle beziehen sich nur auf die Schnittstelle der abstrakten Klasse `LibraryItem`
- Datei hat gleiche Syntax für alle Domänen (nur die Inhalte variieren)
- die Artikel werden beim Einlesen der Datei als Objekte erzeugt
 - aber der Dateileser muss doch die Konstruktoren der Objekte kennen, oder was?

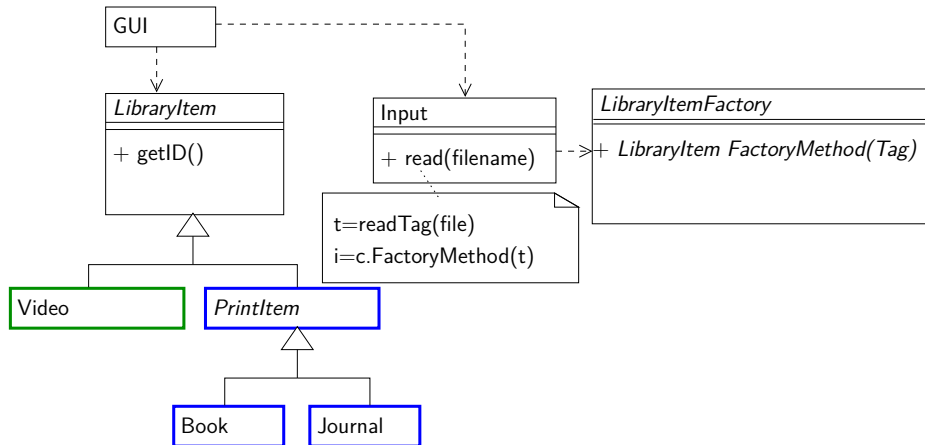
Lösungsstrategie



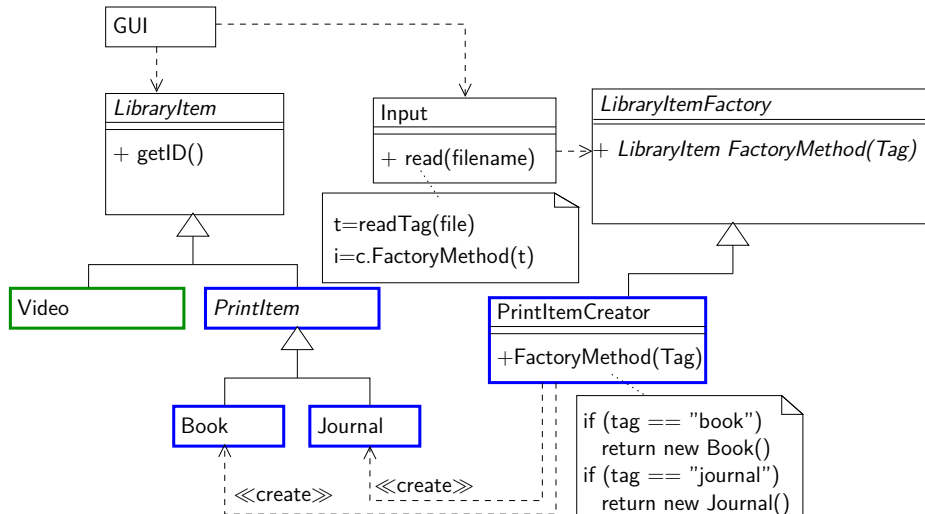
Lösungsstrategie



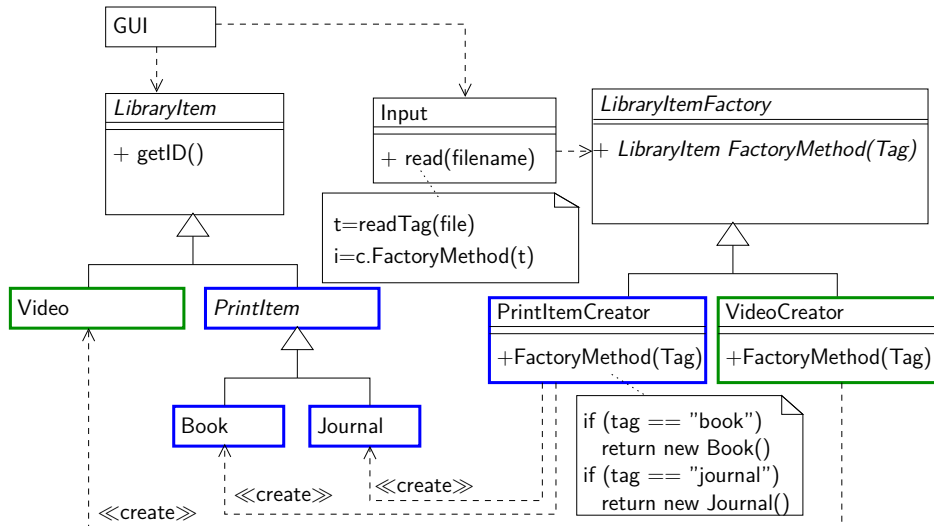
Lösungsstrategie



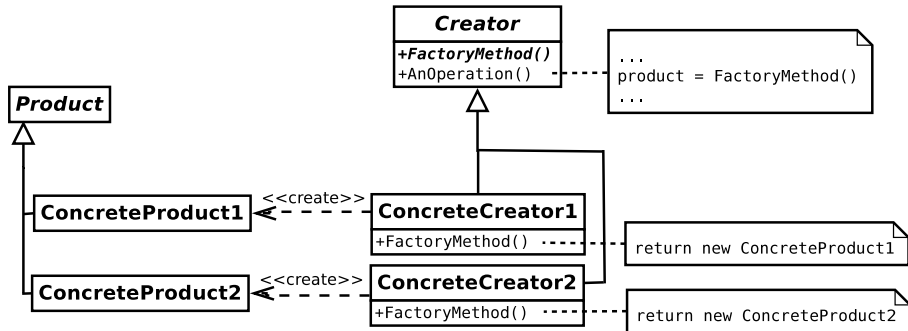
Lösungsstrategie



Lösungsstrategie



Entwurfsmuster: *Factory Method*



LibraryItem-Klassen

```
public abstract class LibraryItem {  
    public String getID() { ... }  
}  
  
public abstract class PrintItem extends LibraryItem {...}  
  
public class Book extends PrintItem {...}  
  
public class Journal extends PrintItem {...}
```

Leser

```
public class Input {  
  
    private LibraryItemFactory creator;  
  
    public Input(LibraryItemFactory creator) {  
        this.creator = creator;  
    }  
  
    private Tag readTag(FileInputStream in) {...}  
  
}
```

Leser (Forts.)

```
public void read(String filename) throws java.io.IOException {
    FileInputStream in = new FileInputStream(filename);
    Tag tag;
    LibraryItem item;
    while (in.available() > 0) {
        tag = readTag (in);
        try {
            item = creator.FactoryMethod(tag);
        }
        catch (LibraryItemCreator.UnknownTag e) {
            System.out.print("unknown_tag_" + tag);
            // keep going
        }
    }
    in.close();
}
```


Creator

```
public abstract class LibraryItemCreator {  
  
    public class UnknownTag extends Exception {};  
    abstract LibraryItem FactoryMethod(Tag tag) throws UnknownTag;  
  
}  
  
public class PrintItemCreator extends LibraryItemCreator {  
  
    PrintItem FactoryMethod(Tag tag) throws UnknownTag {  
        if (tag.equals("book")) {  
            return new Book();  
        } else if (tag.equals("journal")) {  
            return new Journal();  
        }  
        throw new UnknownTag();  
    }  
  
}
```